yagwr

Release 0.0.0

Pablo Yanez

Nov 24, 2021

CONTENTS:

CHAPTER

ONE

INSTALLATION

To install this package:

```
pip install yagwr
```

CHAPTER

TWO

USAGE

After the installation, a script called `yagwr` will be available:

```
yagwr rules_and_actions.yml
```

For a complete list of all command line options, please execute:

```
yagwr --help
```

2.1 Default host & port

By default, `yagwr` connects to `127.0.0.1` and listens on port `7777`. Use the `--host` and `--port` options to change this values.

2.2 SSL support

`yagwr` has no native SSL support. It is recommended that you use NGINX or Apache and configure a reverse proxy.

2.2.1 Reverse proxy with `NGINX`

To setup reverse proxy with `NGINX`, you need to do the following:

```
server {
    listen 443 ssl;
    server_name subdomain.domain.tld;

    ssl on;
    ssl_certificate     /etc/letsencrypt/live/subdomain.domain.tld/fullchain.pem;
    ssl_certificate_key /etc/letsencrypt/live/subdomain.domain.tld/privkey.pem;

    access_log  /var/log/nginx/ssl_subdomain.domain.tld-access.log;
    error_log   /var/log/nginx/ssl_subdomain.domain.tld-error.log;

    location / {
        proxy_cache off;
        proxy_pass  http://localhost:7777;
        include /etc/nginx/proxy_params;
```

(continues on next page)

(continued from previous page)

```
        proxy_read_timeout 3600;
    }
}
```

Note: On Debian based operating systems the file `/etc/nginx/proxy_params` is usually present. If that's not the case, then create this file with this content:

```
proxy_set_header Host $http_host;
proxy_set_header X-Real-IP $remote_addr;
proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
proxy_set_header X-Forwarded-Proto $scheme;
```

See also: https://docs.nginx.com/nginx/admin-guide/web-server/reverse-proxy

2.2.2 Reverse proxy with Apache

To setup reverse proxy with `Apache`, you need to do the following:

```
<VirtualHost *:443>
    ServerName subdomain.domain.tld

    ErrorLog ${APACHE_LOG_DIR}/subdomain.domain.tld-error.log
    CustomLog ${APACHE_LOG_DIR}/subdomain.domain.tld-access.log combined

    SSLEngine on
    SSLCertificateFile      /etc/letsencrypt/live/subdomain.domain.tld/cert.pem
    SSLCertificateKeyFile   /etc/letsencrypt/live/subdomain.domain.tld/privkey.pem
    SSLCertificateChainFile /etc/letsencrypt/live/subdomain.domain.tld/fullchain.pem

    ProxyPreserveHost on
    ProxyPass / http://127.0.0.1:7777/
    ProxyPassReverse / http://127.0.0.1:7777/
</VirtualHost>
```

2.3 Actions & rules

`yagwr` parses a YAML file that contains rules and actions. When Gitlab sends a POST request to the server, `yagwr` goes through the list of rules. If a rule matches the request, then the action is executed.

2.3.1 Format

The top level structure of the YAML file is a list with this shape:

```
---

- condition: <COND>
  action: <ACTION>

- condition: <COND>
  action: <ACTION>

...
```

The file must have at least one condition.

2.3.2 Rules (<COND>)

The following request properties can be checked in the rules:

Property	Description
`path`	The request path, e.g. `/webhook`
`gitlab_token`	Value of the `X-Gitlab-Token` header
`gitlab_event`	Value of the `X-Gitlab-Event` header
`gitlab_host`	Hostname of the gitlab instance take makes the request

The condition can be either

- `key` <OP> `value` where `key` is a property as shown in the table above and <OP>:
  - =: equals
  - !=: not equals
  - ~=: match regular expression
  - !~=: does not match regular expression
- `any: LIST of conditions`: at least one condition must be true
- `all: LIST of conditions`: all conditions must be true
- `not: condition`: negates the condition

Examples

- `X-Gitlab-Event` *must be 0xdeadbeef*:

```
- condition: gitlab_token = 0xdeadeef
```

- `X-Gitlab-Event` *must be 0xdeadbeef and the host must match gitlab[0-9]+.example.com*:

```
- condition:
        all:
            - gitlab_token = 0xdeadbeef
            - gitlab_host ~= gitlab[0-9]+.example.com
```

- `X-Gitlab-Event` *must be either* Push Hook *or* Tag Push Hook and the host must not be invalid.example.com

```
- condition:
        all:
          - any:
              - gitlab_event = Push Hook
              - gitlab_event = Tag Push Hook
          - not:
              - gitlab_host = invalid.example.com
```

2.3.3 Actions (<ACTION>)

The string passed in the action is executed using */bin/sh* login shell.

All HTTP-headers sent in the request are exported as environment variables with the prefix `YAGWR_` and white spaces and dashes are replaced by underscores. For example the value of `X-Gitlab-Token` is available as the environment variable `YAGWR_X_Gitlab_Token`.

The body of the request is piped into the `stdin` buffer of the first process defined in the action.

The return code of the action is ignored by `yagwr`, however it waits for the action to exit before it continues with the next action.

The action is executed in the same directory where `yagwr` is being executed from.

Examples

```
action: /home/project_a/doc/build_docs.py | sendmail status@mycompany.com
```

2.4 Example

```
---

- condition:
    all:
        - gitlab_token = da89d228826a2ac5ba9abdf438182cfc
          gitlab_event = Push Hook
  action: ~/local/bin/checkout_repo.sh
```

(continues on next page)

(continued from previous page)

```
- condition: path = /a64/logger
  action: python3 ~/local/bin/log_gitlab_event.py > ~/logs/log_gitlab_event.log
```

CHAPTER

THREE

API DOCUMENTATION

3.1 yagwr.async_in_thread: Running an asyncio loop in a thread

yagwr.async_in_thread.**module_logger = <Logger yagwr.async_in_thread (WARNING)>**
The default logger of this module

class yagwr.async_in_thread.**AsyncInThread**(*coro*, *name='AsyncThread'*, *log=<Logger yagwr.async_in_thread (WARNING)>*)

Bases: object

This class allows to execute an asyncio loop in a thread.

Sometimes you need to execute asynchronous code in a seprate thread inside a synchronous program. Starting the ioloop in a thread is a chore. This class allows you to do that.

Inside your main task, you can get the running loop via asyncio.get_running_loop(). The loop will have an extra attribute thread_controller with a reference to the AsyncInThread object.

Example:

```
import asyncio
from yagwr.async_in_thread import AsyncInThread

async def main_task():
    print("This is the main task")
    while True:
        print("Doing stuff")
        await some_other_function()
        await asyncio.sleep(1)

ath = AsyncInThread(main_task())

ath.start()
try:
    while True:
        execute_task()
        if should_quit():
            break
finally:
    ath.stop()
```

Parameters

- **coro** (*coroutine*) – a coroutine, the main task. When *stop()* is executed, the task is cancelled. The task is responsible to cancel other tasks that it might have spawned.
- **name** (*str*) – a string used in logging and for the name of the thread
- **log** (*logging.Logger*) – the logger where debug info is logged to.

start()

stop()

3.2 yagwr.checker: The condition checker

This module implements a simple checker that checks whether a condition matches in a dictionary. It is similar to JsonLogic but it is also much more simpler because it's not a general solution and only matches dictionaries whose keys and values are strings only.

With this module you can solve *Is the value of key A equals 5 and does this regex match the value of key B?*-kind of questions.

The conditions can be built directly by generating *Node* objects and linking them toghether according to your logic rules, or you can create a dictionary and parse it with *parse_from_object()*.

3.2.1 The condition dictionary

You have three basic operators: **ANY** (corresponds to boolean `OR`), **ALL** (corresponds to boolean `AND`) and **NOT** (corresponds to boolean `NOT`).

The basic grammer rules are:

```
<node> ::= <terminal-node> | <OP>([<node>, <node>, ...])

<terminal-node> ::= "key = value" | "key != value" | "key ~= regex" | "key !~= regex"

<OP> ::= "ANY" | "ALL" | "NOT"
```

The <OP> (operator) corresponds to the dictionary key. The operands (the other <node>s) are decoded inside a list. That means that you always need at least one operator.

3.2.2 Example

We want to implement this condition:

```
(akane != kun) OR ( (genma = san) AND (nabiki ~= tendou?) )
```

The dictionary with this rules is:

```
{
    "ANY": [
        "akane != kun",
        {
            "ALL": [
                "genma = san",
```

(continues on next page)

(continued from previous page)

```
                "nabiki ~= tendou?"
            ]
        }
    ]
}
```

The following dictionary will match the condition:

```
{
    "akane": "chan",
    "ranma": "kun",
    "genma": "san",
    "nabiki": "tendo",
}
```

The following dictionary will **not** match the condition:

```
{
    "akane": "chan",
    "ranma": "kun",
    "genma": "saotome",
    "nabiki": "tendo",
}
```

Note: For simplicity, the left-hand-side of `<terminal-node>` string supports letters, numbers, dashes and underscores only. The module uses the following regular expression `\w[\w\s]*` to match the left-hand-side.

Adding full unicode support would make the code unnecessarily complicated, specially since in `yagwr` the dictionaries to be matched are going to contains those characters only.

If you need something more powerful or a more general solution, we recommend JsonLogic.

---

exception yagwr.checker.**InvalidExpression**

Bases: Exception

This exception is raised when parsing the condition-dictionary fails because of an incorrect type was passed.

class yagwr.checker.**Node**(*kind*, *children=[]*)

Bases: object

The base node. All nodes must have at least one children.

Do not instantiate this class directly.

Parameters

- **kind** (*str*) – A string representation of the kind of the node
- **children** (*list*) – a list of the children of the node, they must be of type *Node*.

eval(*ref*)

Evaluates the condition in the node given a dictionary

Parameters **ref** (*dict*) – the dictionary to be evaluated

Returns bool `True` if the condition matches the values in the dictionary, `False` otherwise.

`to_dict()`

Returns The condition in dictionary form.

Return type dict

`class yagwr.checker.LiteralNode(expr)`
Bases: *yagwr.checker.Node*

A *Literal Node*, that means it's a terminal node. It doesn't have children.

Parameters **expr** (*str*) – the boolean expression. The operator can be one of: = (equals), != (not equals), ~= matches regular expression, !~= doesn not match regular expression. The left-hand-side and the right-hand-side values are trimmed.

`eval(ref)`
Evaluates the condition in the node given a dictionary

Parameters **ref** (*dict*) – the dictionary to be evaluated

Returns bool `True` if the condition matches the values in the dictionary, `False` otherwise.

`class yagwr.checker.NotNode(node)`
Bases: *yagwr.checker.Node*

A **NOT** *Node*.

Parameters **node** (Node) – The node to be negated

`eval(ref)`
Evaluates the condition in the node given a dictionary

Parameters **ref** (*dict*) – the dictionary to be evaluated

Returns bool `True` if the condition matches the values in the dictionary, `False` otherwise.

`class yagwr.checker.AllNode(nodes)`
Bases: *yagwr.checker.Node*

A **AND** *Node*.

Parameters **nodes** (*list(*Node*)*) – A list of nodes that all must individually match the condition.

`eval(ref)`
Evaluates the condition in the node given a dictionary

Parameters **ref** (*dict*) – the dictionary to be evaluated

Returns bool `True` if the condition matches the values in the dictionary, `False` otherwise.

`class yagwr.checker.AnyNode(nodes)`
Bases: *yagwr.checker.Node*

A **OR** *Node*.

Parameters **nodes** (*list(*Node*)*) – A list of nodes. Only one must match the condition.

`eval(ref)`
Evaluates the condition in the node given a dictionary

Parameters **ref** (*dict*) – the dictionary to be evaluated

Returns bool `True` if the condition matches the values in the dictionary, `False` otherwise.

`yagwr.checker.parse_from_object(obj)`
Parses the condition from a dictionary.

Parameters **obj** (*dict*) – The dictionary containing the condition. See *condition dictionary* for the structure of the dictionary.

Returns The node representing the out-most operator of the condition

Return type *Node*

3.3 yagwr.rules: Rules container

class yagwr.rules.**Rule**(*condition*, *action*)

Bases: object

This class reprents a rule. A rule is the cobination of a condition and action. If the condition matches, the action can be executed.

Parameters

- **condition** (checker.Node) – A Node object that holds the condition
- **action** (*any*) – the action. This object just stores the action, it doesn't manipulate it. Hence you can set any object you like.

matches(*obj*)

Return whether the condition matches the object

Parameters **obj** (*dict*) – The object with which the condition is checked. See checker for more information about the shape of the object.

Returns True if the condition matches, False otherwise

Return type bool

classmethod from_dict(*obj*)

Generates a new *Rule* object from a dictionary.

The dictionary must have two key:

- condition: see *The condition dictionary* for more information
- **action: an object (usually string) with the action to be taken** when the condition matches

Returns A new rule

Return type *Rule*

Raises ***checker.InvalidExpression*** – when parsing the condition fails

3.4 yagwr.logger: Logging helpers

class yagwr.logger.**NamedLogger**(*logger*, *extra*)

Bases: logging.LoggerAdapter

A logging adapater that uses the passed name in square brackets as a prefix.

The extra arguments are name, a string. If name is not present or if it's None, no prefix is used.

Initialize the adapter with a logger and a dict-like object which provides contextual information. This constructor signature allows easy stacking of LoggerAdapters, if so desired.

You can effectively pass keyword arguments as shown in the following example:

adapter = LoggerAdapter(someLogger, dict(p1=v1, p2="v2"))

process(*msg*, *kwargs*)
Process the logging message and keyword arguments passed in to a logging call to insert contextual information. You can either manipulate the message itself, the keyword args or both. Return the message and kwargs modified (or not) to suit your needs.

Normally, you'll only need to override this one method in a LoggerAdapter subclass for your specific needs.

exception yagwr.logger.**LoggerConfigError**
Bases: Exception

Exception raised when a configuration error is detected

yagwr.logger.**setup_logger**(*log_file*, *log_level*, *quiet*, *log_rotate=None*, *log_rotate_arg=None*)
Setups the logging based on `log_file`, `log_level`, `quiet`.

Parameters

- **log_file** (*str*) – the log file, `"stderr"` or `"stdout"`
- **log_level** (*int*) – a log level
- **quiet** (*bool*) – If set to `True`, then logging is suppressed
- **log_rotate** (*str*) – the type of rotation, `"time"` or `"size"`
- **log_rotate_arg** (*str*) – rotation arguments

Raises *LoggerConfigError* – when the settings are incorrect

3.5 yagwr.webhooks: Webhooks

class yagwr.webhooks.**WebhookHandler**(*request*, *client_address*, *server*)
Bases: http.server.BaseHTTPRequestHandler

The request handler for Gitlab webhooks

The Gitlab documentation states:

> Your endpoint should send its HTTP response as fast as possible. If the response takes longer than the configured timeout, GitLab assumes the hook failed and retries it.

For this reason this request handler pushes the request information (headers, payload) onto a `asyncio.Queue` queue and responds as fast as possible. This approach is fine because the documentation also says:

> GitLab ignores the HTTP status code returned by your endpoint.

Hence it doesn't matter whether the processing takes a long time or even fails.

The processing itself is executed in a `asyncio` task.

finish_request()
Helper that finished the request

do_POST()
Handles the HTTP request from gitlab

log_message(*fmt*, **args*)
Log an arbitrary message.

This is used by all other logging functions. Override it if you have specific logging wishes.

The first argument, FORMAT, is a format string for the message to be logged. If the format string contains any % escapes requiring parameters, they should be specified as subsequent arguments (it's just like printf!).

The client ip and current date/time are prefixed to every message.

async yagwr.webhooks.**process_gitlab_request_task**(*controller*)
Main asyncio tasks that reads the requests from the queue and launches the processing of the queue

async yagwr.webhooks.**execute_action**(*request*, *action*, *log*)
Helper function that executes arbitrary commands

CHAPTER

FOUR

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

y

INDEX